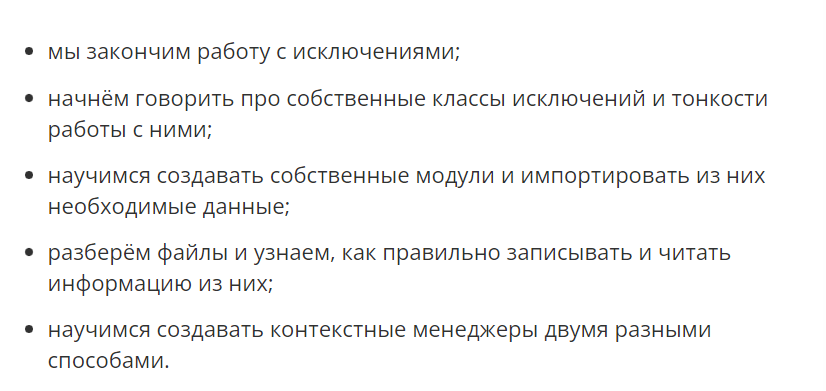
**Модуль B8. Модули и импорт. Работа с файлами и данными.**

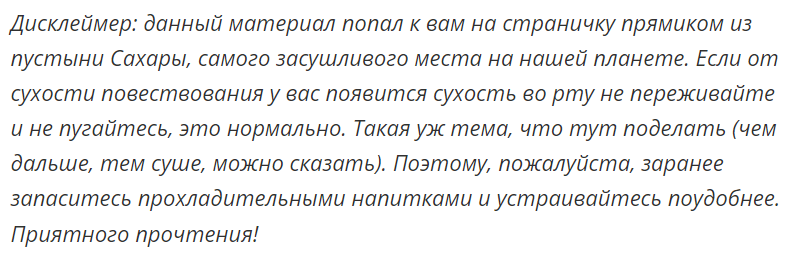
Ладно, вернемся к Python. С гитом конечно надо будет еще по экспериментировать.

Планы.



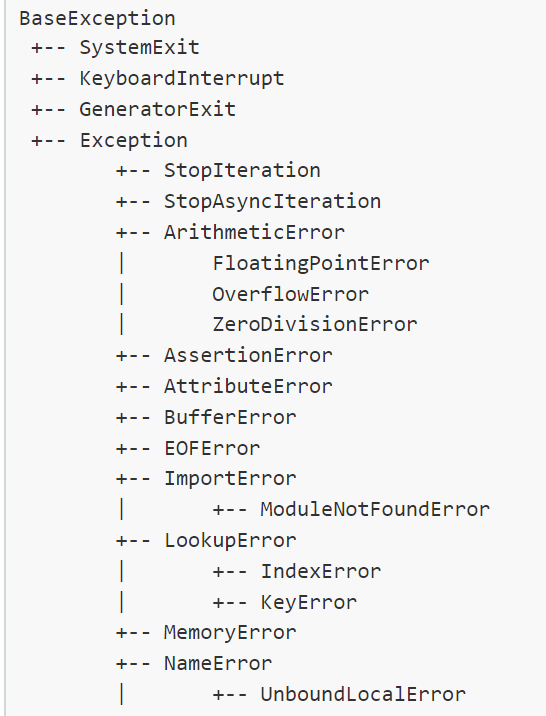
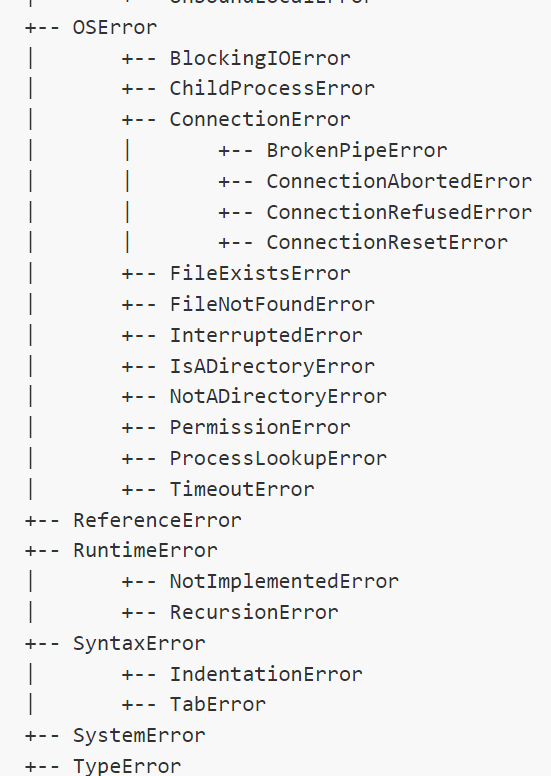
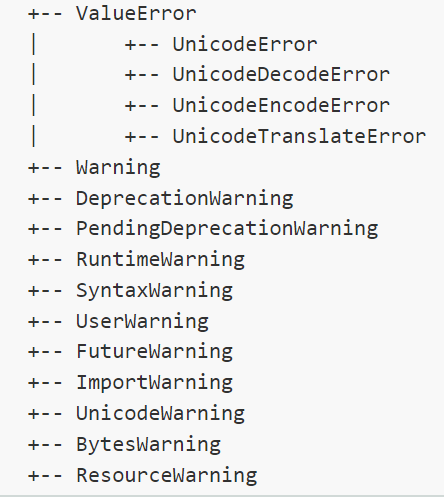
Чтение и запись в файлы это особенно хорошо.

**B8.2. Тонкости обработки исключений. Собственные классы исключений.**



Забавная шутка. Оценю сухость от 0 до 10.

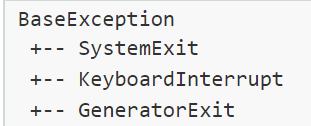
Попробуем написать свой класс исключения. Но сначала какая-то схема:

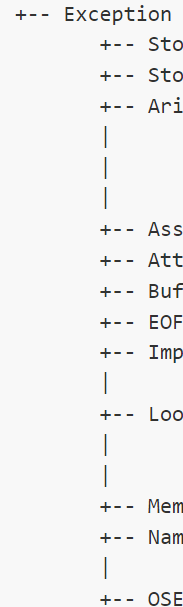
Эта система — **дерево стандартных исключений**.

Вся эта куча это стандартные исключения, которые наследуются от класса **BaseException.**

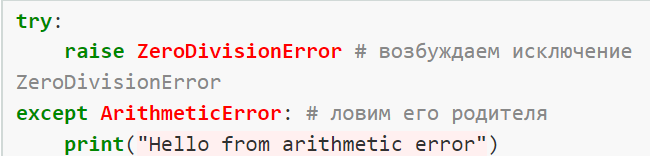
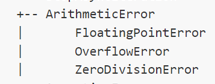
**SystemExit, KeyboardInterrupt**, **GeneratorExit** это исключения, которые нельзя отловить. Причина в том, что их возникновение не зависит от выполнения программы.



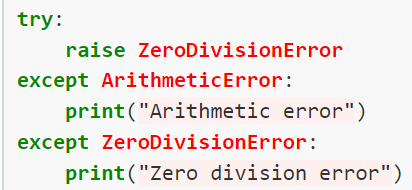
А вот все остальное, что наследуется от класса **Exception,** можно**.**



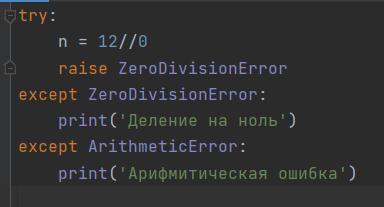
Хотя некоторые из них и возникают крайней редко. Важно понять, что в блоке except можно ловить не только сам класс, но и его родителя:

Отлавливать ошибки можно целыми списками. Но важно соблюдать структуру. К примеру вот пример:

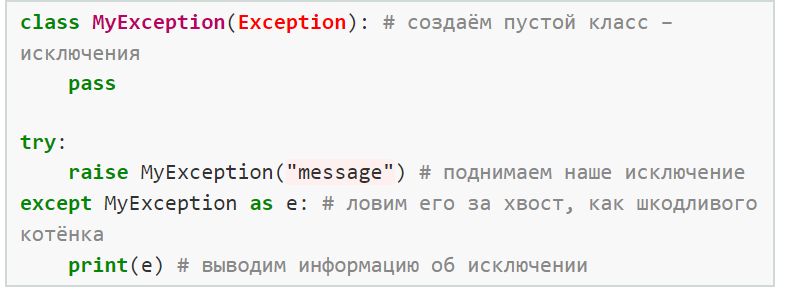


Вывод будет: **Arithmetic error.** Дело в том, что **ArithmeticError** является родителем **ZeroDivisionError.** В и хотя ошибки и ZeroDivision, но т.к. выше отловлена его ошибка, то выводится будет это исключение. Попробую

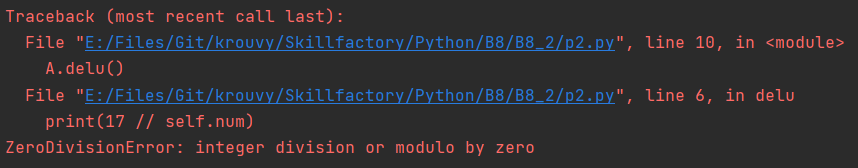
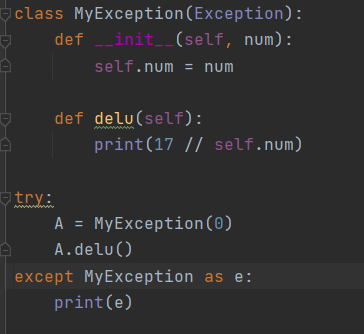


То есть если нужно отловить конкретный класс, то исключение этого класса нужно прописать выше тех, что находятся выше его по иерархии.

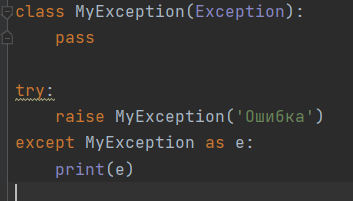
Потом можно будет писать свои исключения. Но мне пока не совсем ясно как они будут выглядить.



Ну то есть, что должно произойти MyException… Хм.. а если я внутри напишу функцию деления на ноль.

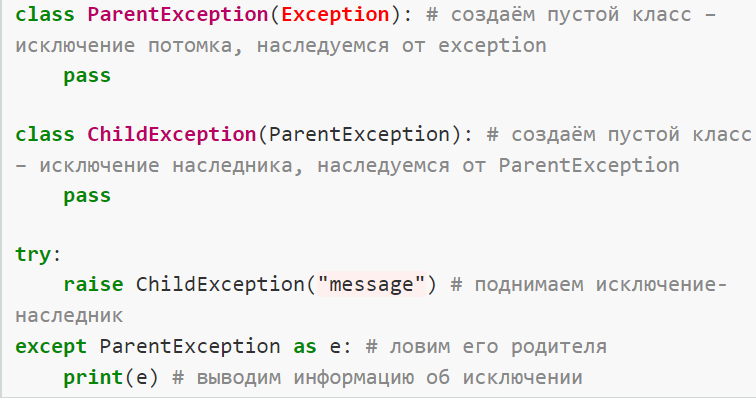


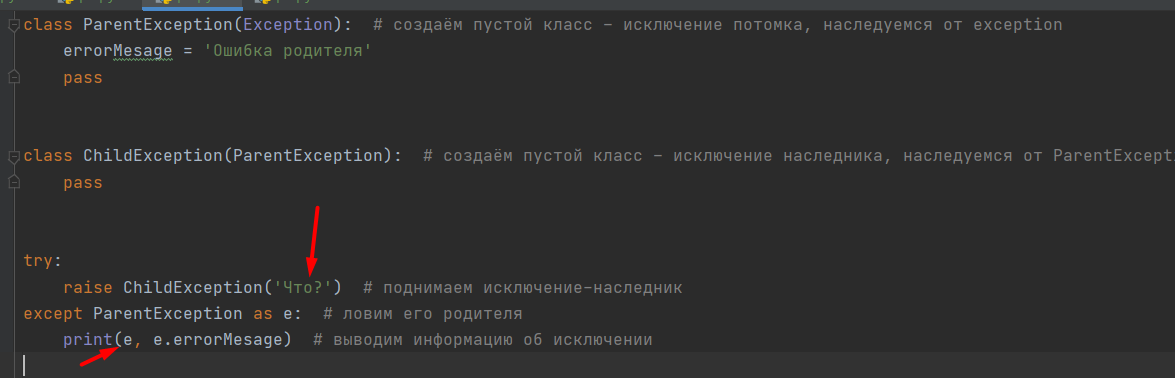
Ну повторить код не получилось. Чего-то не хватает в таком примере. А с таким пример лично мне не понятно. Это же имитация простая.



Желательно чтобы исключения наследовались друг от друга, но в первую очередь от класса Exception. Наследование нужно вести, чтобы в итоге можно было отлавливать отдельные исключения, касающиеся ресурсов, к примеру ОЗУ.

Ну вот пример. С наследованием и ловлей родителя. Но опять же. raise это имитация.



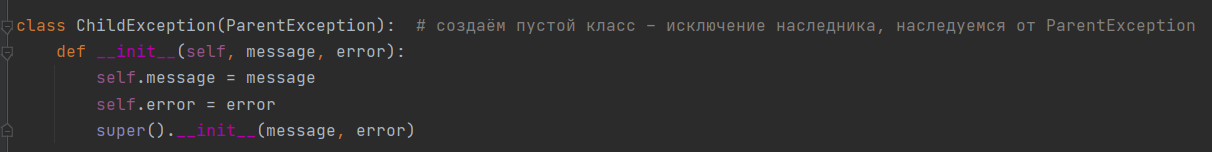


Мне не понятна одна часть. Чем является аргумент ChildException. Не понимаю, почему этот класс, вообще что-то принимает. Причем Когда доходит до обработки исключения. То “e” является одновременно и сообщением из ChildException и самим классом родителя, таким образом можно получить доступ к его переменным:

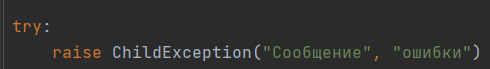


Но отлов происходит. Хотя отлавливали родителя. Можно и в принципе заменить на Except и он тоже отловится. Но если в родителе убрать наследование от Exception?

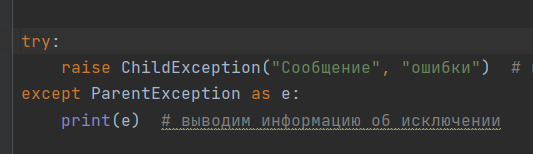
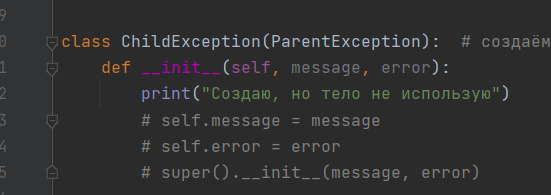
Разберу код по частям.



Вот дочерний класс с инициализацией. Его можно вызвать отловив ошибку его родителя:



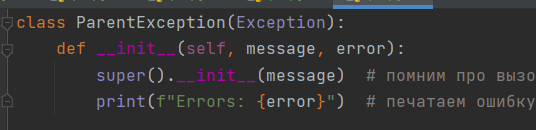
Но если убрать из инициализации 3 этих строки:



То в except мы получим кортеж, который передали.

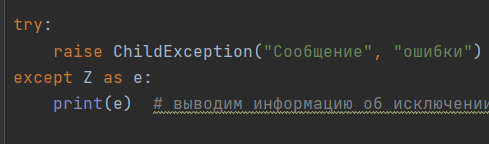
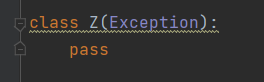


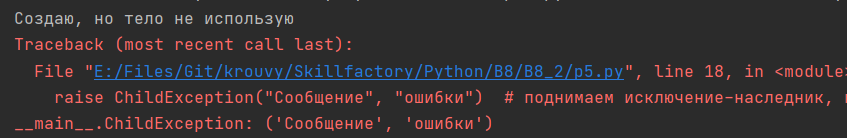
Даже учитывая, что класс исключения прописан:



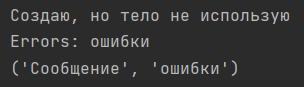
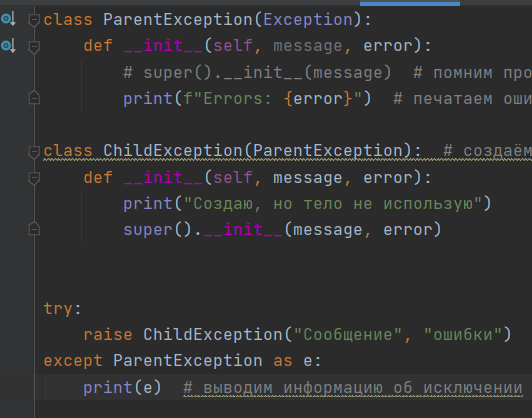
Но проблема в том, что без super у него инициализация не прошла, но тем не менее они связаны наследством. Поэтому как-то, но Exception отработал. Но если бы они не были связаны, скажем так:

То будет ошибка:

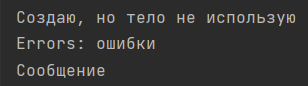




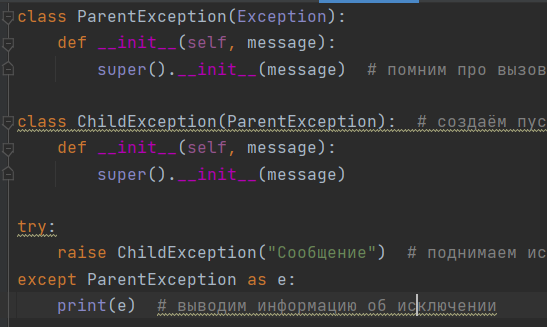
Ладно разрешу инициализировать класс родителя в дочернем классе через Super. Но в родителе инициализировать Exception не дам. Что получу?

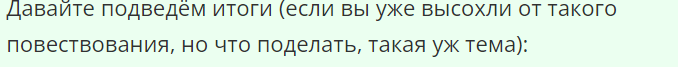


Видно в родителя передалось error и с его помощью вывело строку. Но внутри except print все еще выводит кортеж e. Если убрать комментарий со строки super у родителя, то родитель инициализирует экземпляр класса Exception, с переданным сообщением. В таком случае внутри Except будет передано именно это сообщение, вместо кортежа.

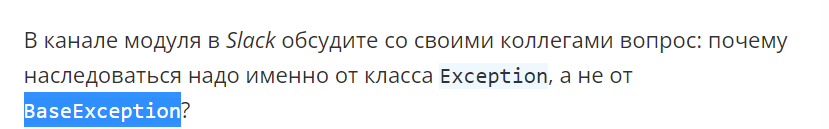


Если так посмотреть, то родителю можно вовсе не передавать **error**.

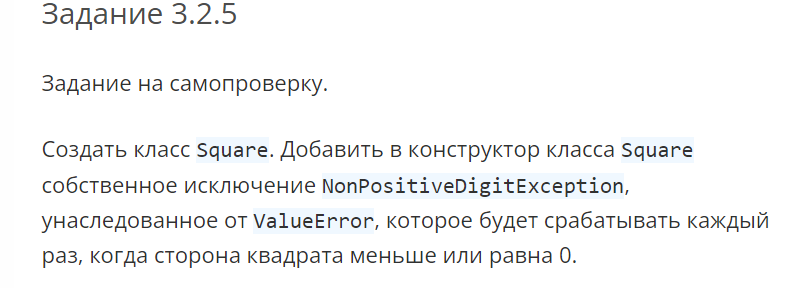




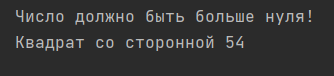
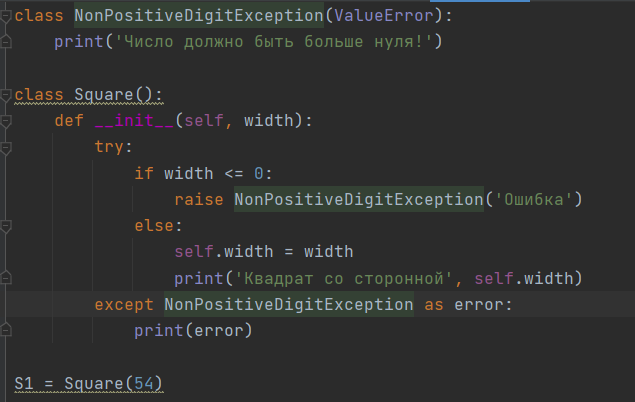
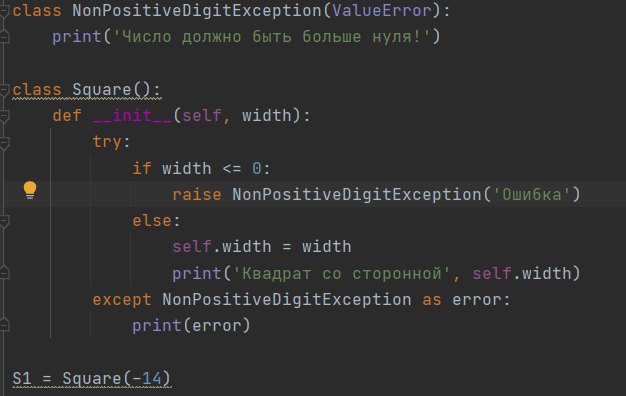
Нет не высох, но опять вы нихера не объяснили. Но видимо на таких примерах пока никак.



Потому что в BaseException есть 3 не отлавливаемых в программе исключения.



Вот блять нормальный пример. Попробую. Не получилось:

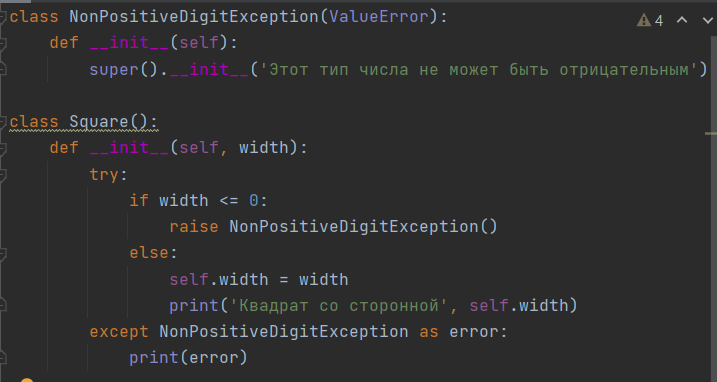


Ну по крайней мере не совсем получилось. Присвоение у меня не происходит. Если условие отрабатывает. Но почему что-то даже если срабатывает, все равно класс NonPositiveException выводит сообщение.

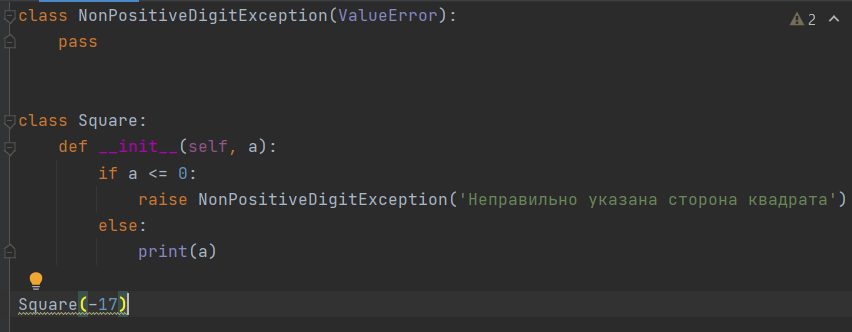
А понял, оно сразу срабатывает, нужна инициализация:

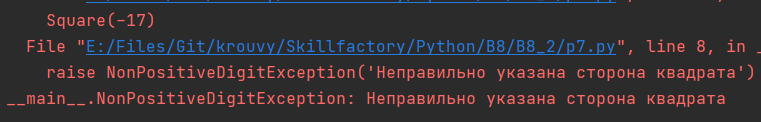
 



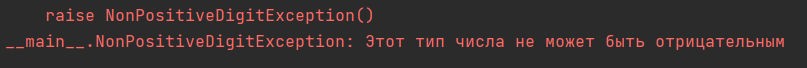
**А теперь по хз какому тз, посмотрим как на самом деле нужно было делать.**





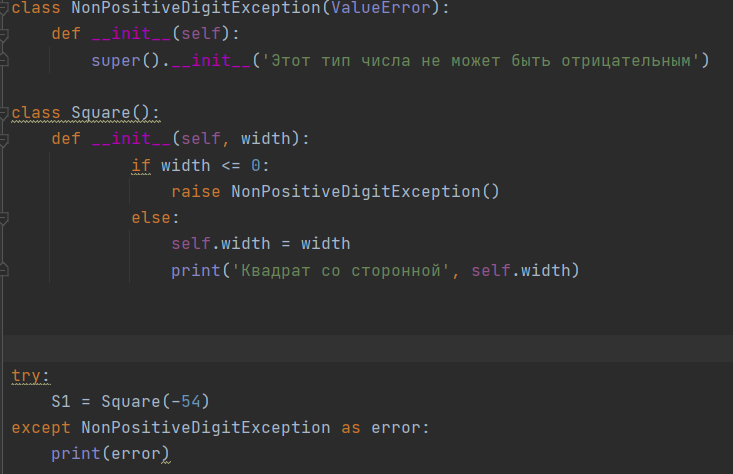
Ясно требовалось создать вызов исключения, а не его обработку. В остальном ок.

У меня если убрать try, тоже видно:

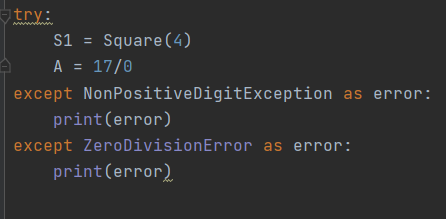


Ладно, стало чуть понятнее как использовать это в жизни. Но не до конца.

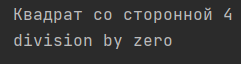
Уху!! Придумал как это норм делать. Не в классе! Я про обработку исключения.



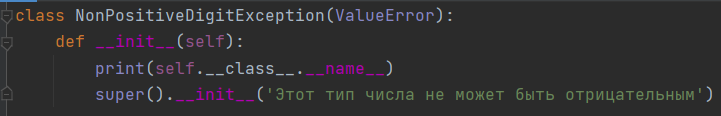
Само создание класса в try обернуть. И здесь если что можно другие ошибки обернуть.

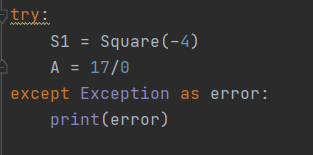


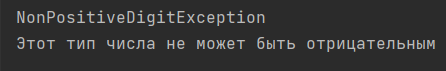
Типо такого



И если не решать проблемы, то можно постепенно их ловить. К примеру в классах исключениях прописывать имя класса, кроме ошибки.



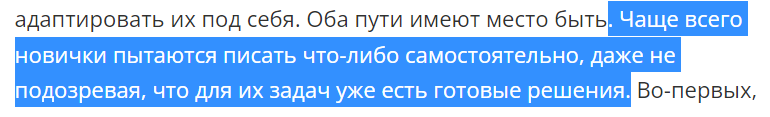




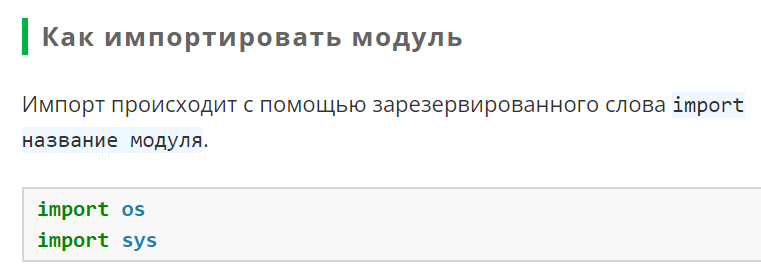
Ну душным или сухим я бы не назвал. Важны все мелочи.

**B8.3. Работа с импортом**

В импорты честно совсем чуть-чуть тыкал. Может узнаю важные вещи понятным языком. Но импорт из папок другого уровня был по настоящему никаким. Ну то есть буквально. У меня не получилось (

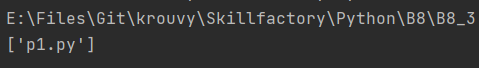


Ну это не плохо, если самостоятельно можешь что-то написать. Но Согласен, если что-то уже готово, то нет ничего плохого в том, чтобы это использовать. Но использовать и подключать нужно правильно. Как тут написано, не стрелять пушкой по воробьям. Тут полагаю имеется ввиду, то что если нужна какая-то стороння функция, не обязательно для этого импортировать всю библиотеку. Достаточно импортировать саму функцию.



Да помним, помним. Причем тут скажем уже скачанные или встроенные модули.

Вот пример функций модуля **os** c выводом текущей директории и списка файлов этой директории.



Причем вторая функция в качестве аргумента может принять адрес. И скажем если передать туда адрес на уровень выше, то:





Видно, что в этом списке открыт файл, а еще есть 2 папки.

Документацию на модуль можно найти в интернете. Но если он недоступен, но модуль скачан, то можно воспользоваться командой **help(модуль),** должна вывестись документация.

Есть еще dir(), но он выводит все атрибуты.







Но help удобнее, полноценная справка. Каждый модуль нужно импортировать с новой строки с отдельными import, это важно. Хотя работать будет и в одну строку.

Чтобы импортировать не весь модуль, а только функции. То:



Также для импорта существует несколько различных способов записи, но делают они тоже самое: импорт всего модуля и обращение к функции через имя модуля:

















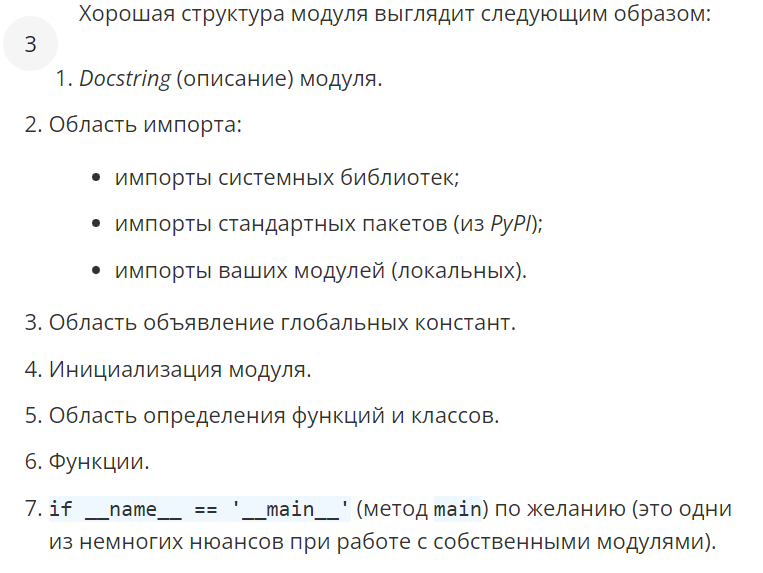
Одно и тоже. Выбирать нужно в зависимости от ситуации.

Как правильно составить модуль?

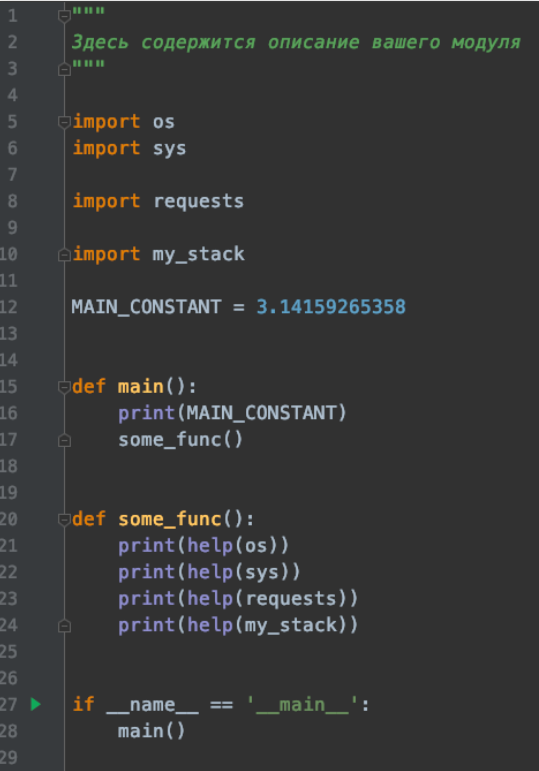
1. Вся логика заключена в отдельные функции, никаких printov вне. На глобальном уровне можно заводить только константы.
2. Если какой-то код должен выполняться только когда модуль не импортирован, то этот код можно обернуть в конструкцию:



Как вариант можно для этого записать отдельную функцию и вызывать ее в таком блоке. Просто напомню что это нужно чтобы чтобы во время импорта эти строки не отрабатывали самостоятельно. Просто если их не обернуть, то они будут отрабатывать даже если импортировать не весь модуль, а только одну функцию.

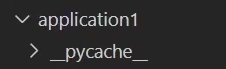
1. 

Пример



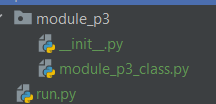
Есть небольшое видео на тему. Глянем

После того, как в какую-то программу импортировали модуль, в папке с этим модулем должен появится файл pycache.

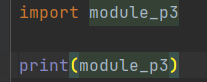
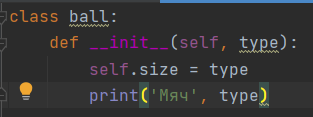


После этого следующий запуск программы будет быстрее, т.к. процесс импорта был ускорен. Так это было интересно. Попробую, хотя и не досмотрел, но вроде усвоил.

Создам маленький модуль. И объясню. Вот структура:



Просто из файла run я, конечно, могу попытаться импортировать модуль. Но…

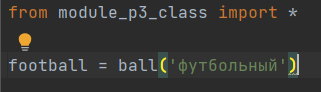


Функционалом даже не пахнет. Для импорта из вложенной структуры нужен файл \_\_init\_\_

Вот что в нем напишем, например.

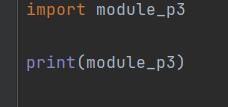


В самом файле init если скажем прописать какой-то объект это будет работать:

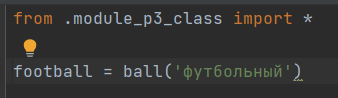




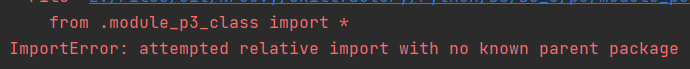
Но если вернутся в файл run будет ошибка.

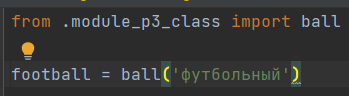
Дело в том, что в init для верности нужно импортировать через точку:

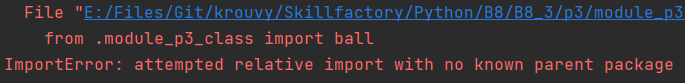


Правда в таком случае сам init не отрабатывает…



Попробую импортировать только один класс, вместо \*

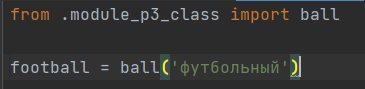




**НЕА**!

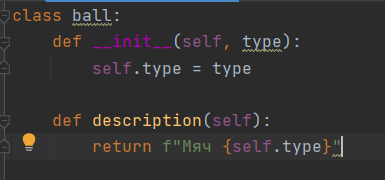
Короче, это не точно. Но видимо если открываем init таким образом, то в нем самом код уже не запустить. Ну так вот что получилось:

В init я делаю импорт и создаю пример объекта.

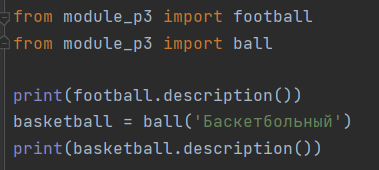


К сожалению таким образом объект будет создавать при каждом обращении к модулю

Но зато можно импортировать себе готовый экземпляр. Будем считать это константой.



Чтобы при каждом импорте не видеть сообщение о футбольном мяче я вынес это в отдельную функцию, описание. А теперь к самому файлу run.



Как видно из модуля свободно экспортируется как класс, так и созданный в init экземляр.

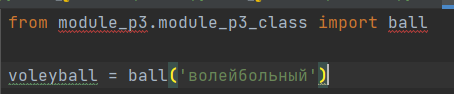
Что еще хорошо. Это что так можно в Run импортировать модули из разных папок, причем сами папки могут взаимодействовать друг с другом. Пример:

Создам еще папку:

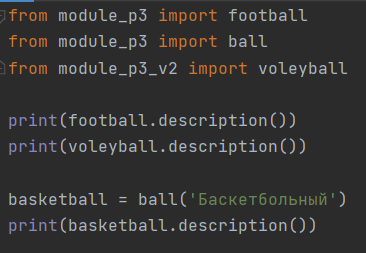


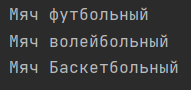
А в ней обращусь к модулю module\_p3 и к файлу module\_p3\_class.

Таким образом в этом init могу создать объект волейбольный мяч.

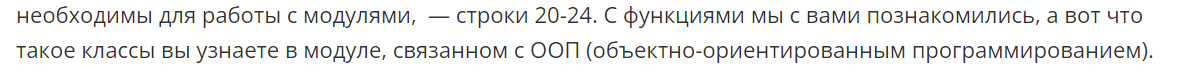


Да, опять же сами init теперь перестают работать. Но зато:

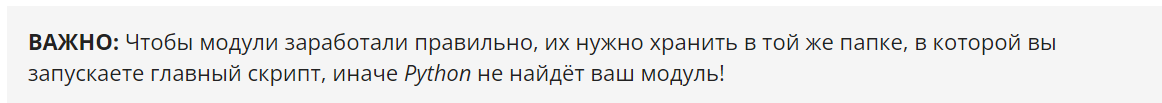




Само все это разобрать куда сложнее. Так что спасибо видео, которое пожалуй стоит и досмотреть.

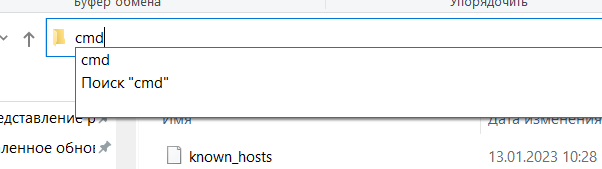


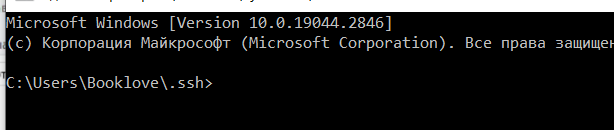
Мда… со структурой у вас не очень, как будто не знали, что ООП разбирали еще в прошлом модуле.



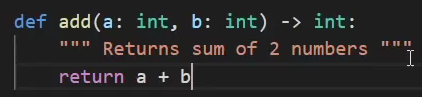
Да, понял.

Чтобы в терминале через **cd** не искать нужный адрес, можно просто в проводнике в адресной строке написать **cmd,** и откроется новый терминал нужным адресом.



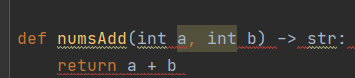


Думаю пока привыкнуть писать функции в стрелочном виде.

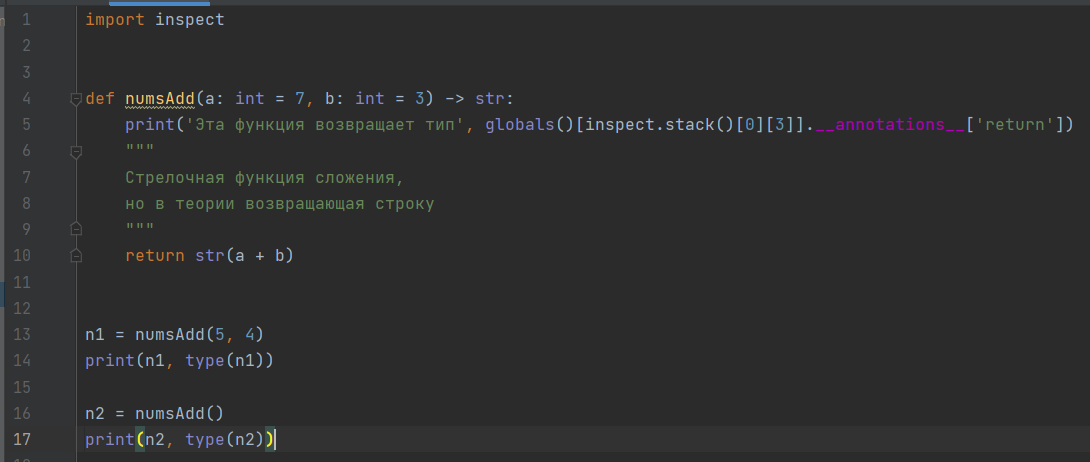


Интересно, если я перепишу эту функцию, но -> сделаю str, то возвращаться будет строковое число?

по памяти я написал так:



Короче по порядку.

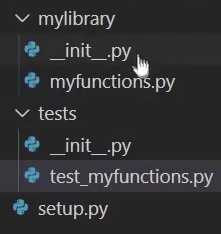


Это все типо документация, которая ни к чему не обязывает, но тон хороший. В скобах описан формат вводимых данных, но это не значит, что программа сразу сломается, если ввести туда другие данные, это просто для удобства понимания. **-> str,** аналогично. Это не значит что из-за этого программа будет возвращать **str,** это просто подсказывает разработчику, что результатом вывода этой программы должен быть этот тип данных. Просто если бы это была более сложная функция, в ней могло быть множество преобразований типов и не обязательно в Return, чтобы за всем этим не следить, можно оставить такую подсказку. А теперь насчет 5-ой строки.

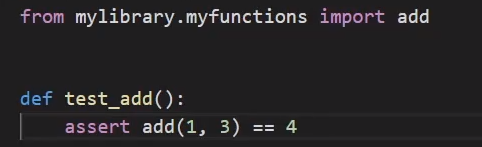
Я уже делал **globals** и **locals**, чтобы обратится к функциям по названию. Я получаю экземпляр этой функции, ну это тоже самое, если бы я написал внутри функции: **numsAdd.\_\_annotations\_\_[‘return’]**

Так вот, с помощью **.\_\_annotations\_\_** и можно получить параметры описания функции. Функция выглядит красивее.

Далее при создании своего модуля не плохо бы ему написать тесты. Обычно это делается в отдельной папке. Вот к примеру структура:

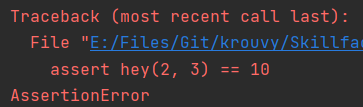
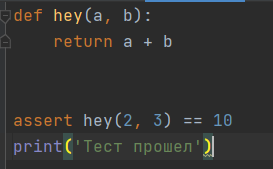


Здесь как видно модуль тестов тоже должен иметь свой init. А внутри теста следующий код:



Сразу хочу выяснить, что такое **assert**

Как я понял, он выводит ошибку если тест не проходит



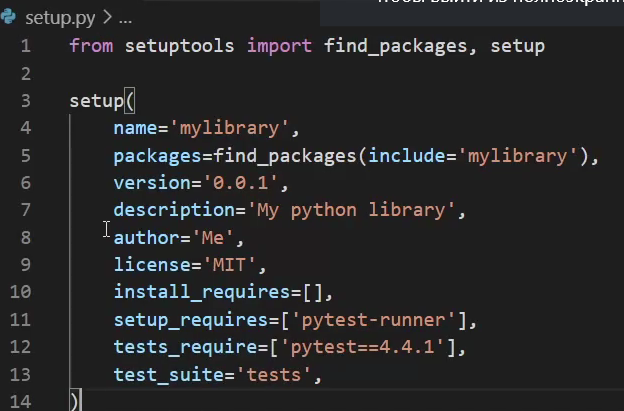


Также можно к **assert** добавить через запятую сообщение, которое будет выводится в случаи ошибки.





А теперь вернемся к файлу **setup.py.** Нем нужно описать следующий код.



**import** это понятно, хотя импортируется именно библиотека **setuptools,** для установки собственных библиотек. Как раз отсюда и импортируется метод **setup.**

**mylibrary,** это имя папки в которой лежит пакет, написанный пользователем. А далее видимо просто идет описание пакета, версия, автор и тд. **license** советуют оставить таким же.

**setup\_requires –** кажется связан с виртуальным окружением. Сюда как я понял добавляются какие-то библиотеки, которые нужны для работы нашего пакета. Если же у этих библиотек требуется какая-то определенная версия для работы, ее можно уточнить в **tests\_require.**

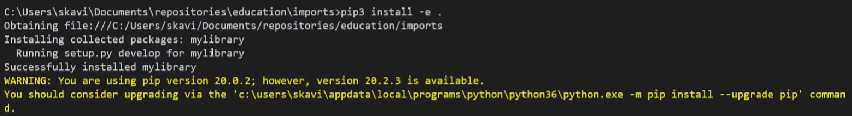
В **test\_suite** как я понимаю прописывается папка с тестами.

И как ее установить теперь?

Для этого нужно будет в проекте прописать

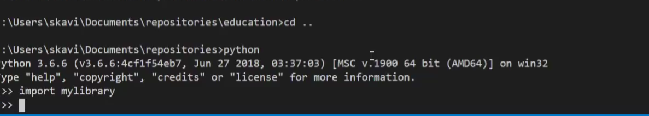
**pip3 install – e .**

Точка в данному случаи нужна потому что мы уже находимся в папке каталога. В противном случае за **точкой** нужно прописать адрес папке и модулем.



Желтый текст это просто сообщение о том, что версия pip не последняя.

И теперь в рамках всего этого проекта можно импортировать этот модуль.

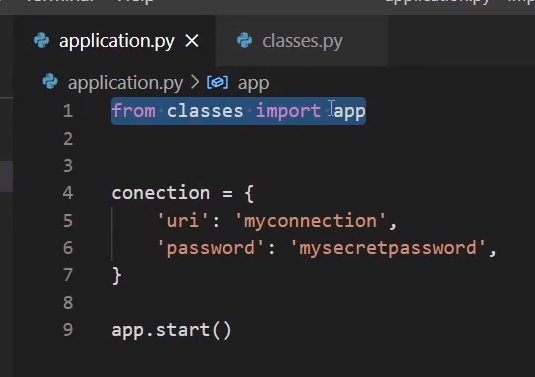
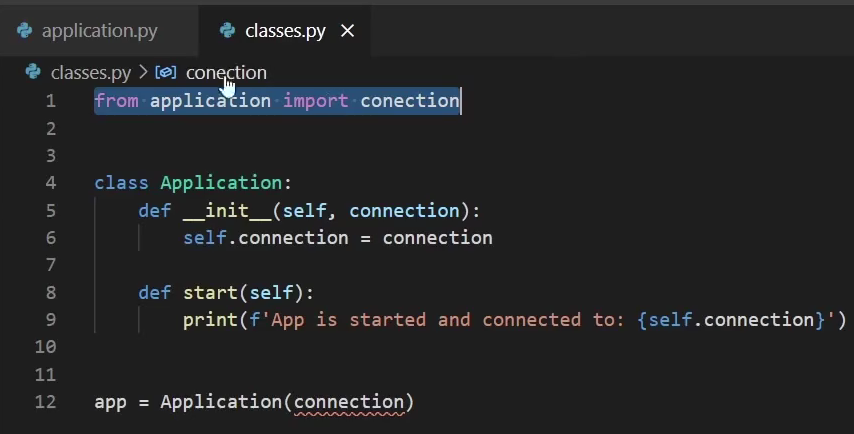


А чтобы ее удалить из проекта:

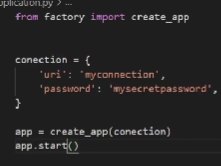
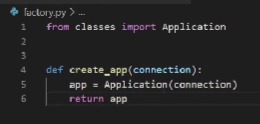


Продолжим. Иногда может случится штука связанная с кросс-импортом. Не знаю о чем речь, но его еще называют перекрестным импортом. В этих случаях вроде разбивают файл на еще более мелкие пакеты. Но если так сделать не выходит. Может помочь шаблон проектирования фабрики. Чуть подробнее о нем:

Кросс импорт это когда допустим в одном файле происходит импорт из другого, а другом из этого. И программа как бы уходит в бесконечный цикл, т.к. пытается все время это сделать:



Достигается максимальная глубина рекурсии. Такая ситуация может возникнуть даже если вынести все в конфиги. Фабрика типо сама создает и собирает приложение.

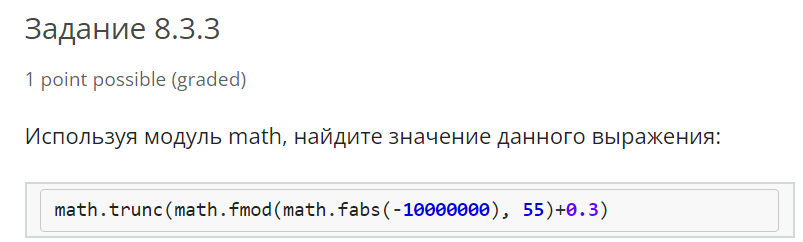


**classes.py** остался неизменным, этот файл может свободно обращаться к application

В **applictation.py** уже обращаемся не к **classes,** а **factory.** А сама **factory** может обращаться к **classes.**

Получается, цельная зависимость. **APP обращается только к factory, classes** только к **application,** а **factory** только к **classes.**

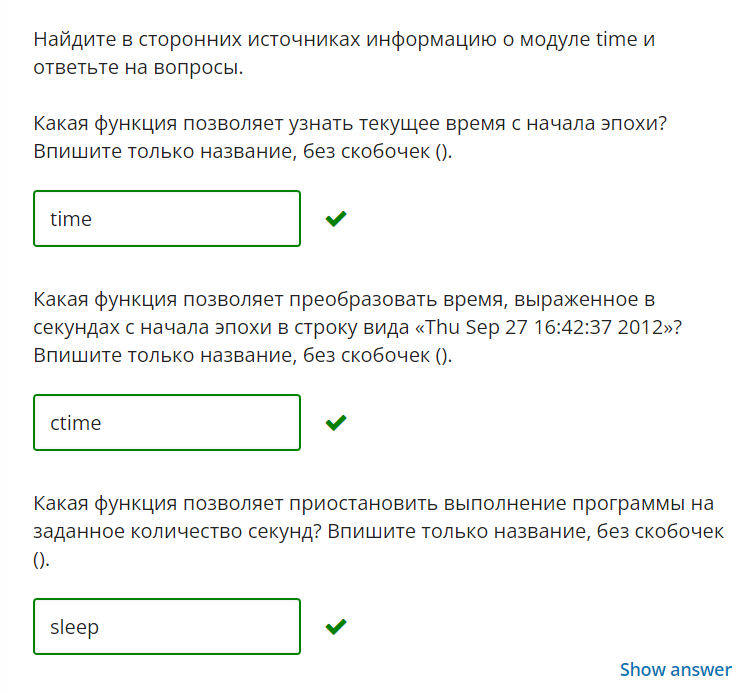
Если честно все равно звучит как замкнутый цикл. Но думаю все верно. С этим хорошо бы поиграться потом.

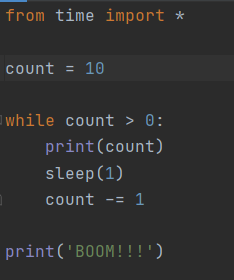
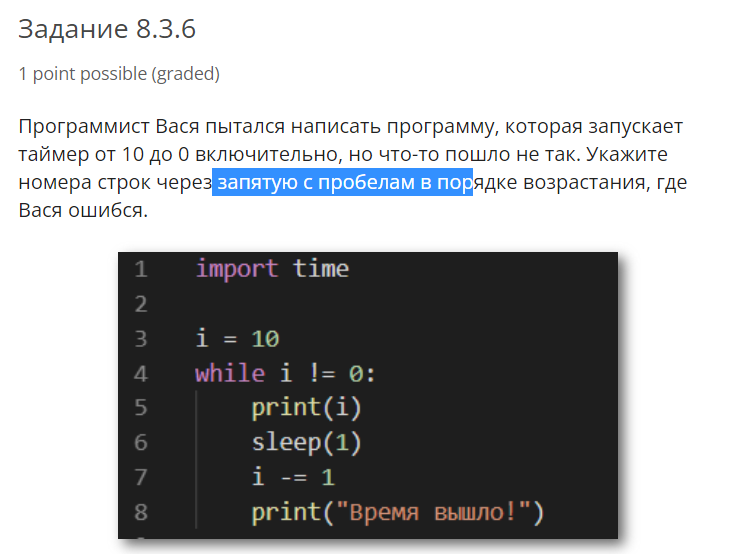


**балин, хер вам. Не буду я эту прогу себе сохранять. Тупо в терминале пропишу.**

Ответ 10

А вот тут немного ошибался. Но все нормуль. Зато запомню лучше.

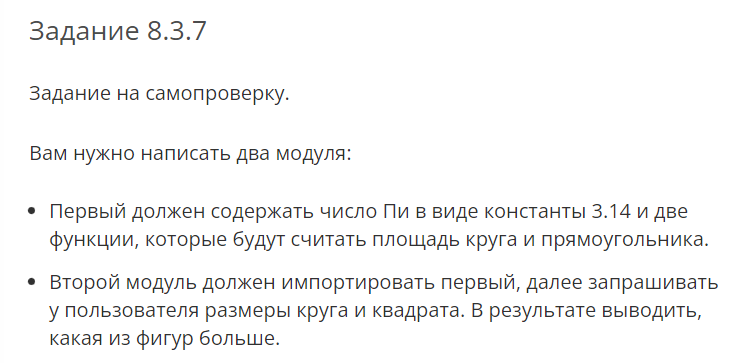




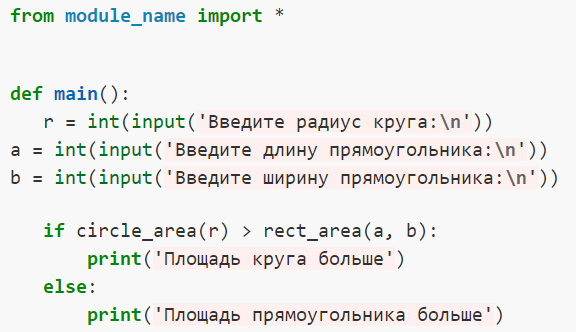
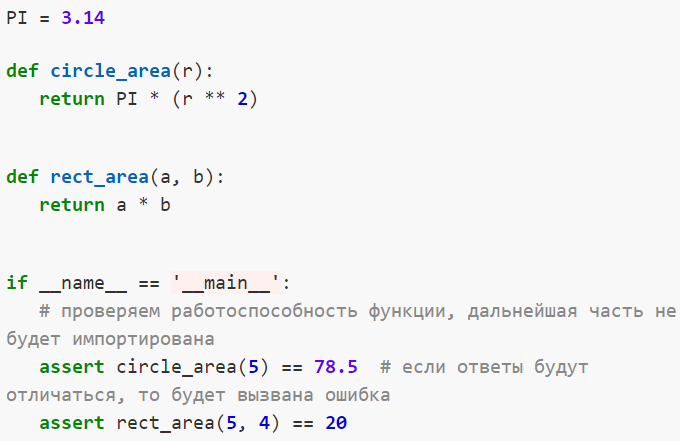
Эй, это так-то интересно. Сам хочу написать.

Было быстро. А у него как вижу ошибки в 4 строке. Знак не тот. И 8-й. Табуляция не там. А еще 6 строка. Он же модуль импортировал не через \*. Так что нужно было слово time.

Это было приятно. Хвалю.



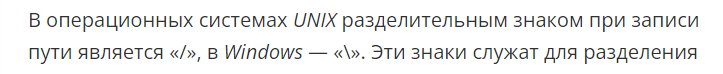
Мне лень. Это скучно. Вот пример ответа.



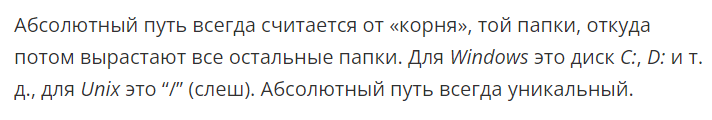
Да, на этой странице много видосов. Долго я на ней просидел. Но тема импортов важна, не могу недооценить ее важность.

**B8.4. Работа с файлами**

А эту тему я хочу разобрать дотошно. В любого файла есть пусть, файла есть путь. И так уж сложилось что чаще всего у этого пути есть слеш. Но то, в какую строну повернут этот слеш зависит от системы.

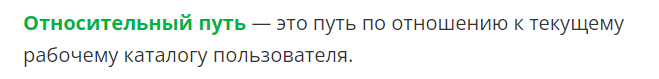


Как и у путей файлов каких-то Css, js и тд, пути бывают **абсолютными** и **относительными**.



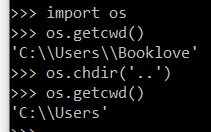
Относительный итак понятно. Что начало берется относительно какого-то места. И этот путь не обязательно будет уникальным, просто для другого файла он будет браться относительно другого места.

Или если проще…



Для работы с этим(в принципе с файлами и путями) принято использовать встроенный модуль **OS.**

Функция **os.chdir()** позволяет изменить текущую директорию. Чтобы узнать текущую директорию можно воспользоваться командой **os.getcwd()**



**os** вообще довольно опасный модуль полагаю. Вирусы без него написать сложно будет полагаю.

**os.listdir() –** файлы текущей директории.



**os.path.join() –** склейка пути с учетом особенности ОС.

Первый аргумент текущий путь, второй аргумент папка.



**Для кроссплатформенности использовать этот метод очень важно.**